Master's Project

Czech
Technical
University
in Prague

F2

Faculty of Mechanical Engineering
Department of Automation
Instrumentation Engineering

Applying Computer Vision, and Machine Learning Techniques for modelling and simulation of autonomous Car

Omar Alif Abelhakim Allam

Supervisor: Oswald Cyril Ing., P.hD.
August 2023

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Allam Omar Alif Abdelhakim**
Personal ID number:	**467842**
Faculty / Institute:	**Faculty of Mechanical Engineering**
Department / Institute:	**Department of Instrumentation and Control Engineering**
Study program:	**Automation and Instrumentation Engineering**
Specialisation:	**Automation and Industrial Informatics**

II. Master’s thesis details

Master’s thesis title in English:

Applying Computer Vision, and Machine Learning Techniques for Modelling and Simulation of Autonomous Car

Master’s thesis title in Czech:

Aplikace metod strojového vidění a strojového učení pro vývoj modelu a simulaci autonomně řízeného vozidla

Guidelines:

Assignment/Task:
1. Research on related machine learning and computer vision techniques for autonomous cars.
2. Research of feasible applications of car driving simulation.
3. Design and Implementation for a model of the self-driving car.
4. Integration and deployment of the trained model within the simulation.
5. Testing of the model in simulation.

Bibliography / sources:

https://www.researchgate.net/publication/275897320_DeepDriving_Learning_Affordance_for_Direct_Perception_in_Autonomous_Driving
https://www.researchgate.net/publication/326223123_LaneNet_Real-Time_Lane_Detection_Networks_for_Autonomous_Driving
https://developer.nvidia.com/blog/deep-learning-self-driving-cars/

Name and workplace of master’s thesis supervisor:

Ing. Cyril Oswald, Ph.D. U12110.3

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **28.04.2023** Deadline for master's thesis submission: **15.08.2023**

Assignment valid until: ______________

Ing. Cyril Oswald, Ph.D.	prof. Ing. Tomáš Vyhlídal, Ph.D.	doc. Ing. Miroslav Španiel, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

[illegible]

Student’s signature

Acknowledgements

I would like to express my gratitude towards my supervisor Ing. Cyril, Oswald, Ph.D. for his expert guidance throughout the thesis. I would also like to thank my family for their unwavering support

Declaration

I hereby confirm that I have independently composed this thesis on the subject of Autonomous Driving and Machine Learning techniques for modeling and simulation of autonomous car. All references and sources utilized have been cited. I possess no compelling objection against the utilization of this thesis in accordance with Section 60 of Act No 121/2000 Sb., as amended

Date & Signature

....

Abstract

In this thesis, we explore the evolving domain of autonomous driving, delving into Machine learning methodologies. We primarily focus on the foundational principles of Deep Neural Networks (DNN) and the specialized architecture of Convolutional Neural Networks (CNN). Our aim was to train a neural network for autonomous navigation within Udacity's Car Simulator environment. The design phase involved meticulous data collection, processing, and image optimization, significantly influencing the model's performance. With a noteworthy validation loss achieved through rigorous optimization, our model showcased substantial robustness. This model was then integrated into a simulation, ensuring real-time bidirectional communication and adept vehicular control. Our research highlights the significant impact that machine learning can have on the development of autonomous vehicles.

Keywords: Autonomous Driving, CNN, DNN, validation loss

Supervisor: Oswald Cyril Ing., P.hD.

Abstrakt

V této práci prozkoumáme vyvíjející se doménu autonomního řízení a ponoříme se do metodik strojového učení. Primárně se zaměřujeme na základní principy Deep Neural Networks (DNN) a specializovanou architekturu konvolučních neuronových sítí (CNN). Naším cílem bylo vytrénovat neuronovou síť pro autonomní navigaci v prostředí Car Simulator společnosti Udacity. Fáze návrhu zahrnovala pečlivý sběr dat, jejich zpracování a optimalizaci obrazu, což výrazně ovlivnilo výkon modelu. Díky pozoruhodné ztrátě ověření dosažené díky přísné optimalizaci náš model předvedl značnou robustnost. Tento model byl poté integrován do simulace zajišťující obousměrnou komunikaci v reálném čase a šikovné řízení vozidla. Náš výzkum zdůrazňuje významný dopad, který může mít strojové učení na vývoj autonomních vozidel.

Klíčová slova: Autonomní řízení, CNN, DNN, ztráta ověření

Překlad názvu: Aplikace metod strojového vidění a strojového učení pro vývoj modelu a simulaci autonomně řízeného vozidla

Contents

Figures

Tables

0.1 List of Abbreviations

NHTSA	National Highway and Traffic Safety Administration
HIL	Hardware-in-the-Loop
SIL	Software-in-the-Loop
MP	Modular Pipeline
IL	Imitation Learning
AI	Artificial Intelligence
ML	Machine Learning
PID	Proportional Integral Derivative
MPC	Model Predictive Controller
CAGR	Compound Annual Growth Rate
ADAS	Advanced Driver Assistance Systems
MSE	Mean Squared Error
MLP	Multiple Layer Perceptron
ANN	Artificial Neural Network
DNN	Deep Neural Network
CNN	Convolutional Neural Network

Chapter 1

Introduction

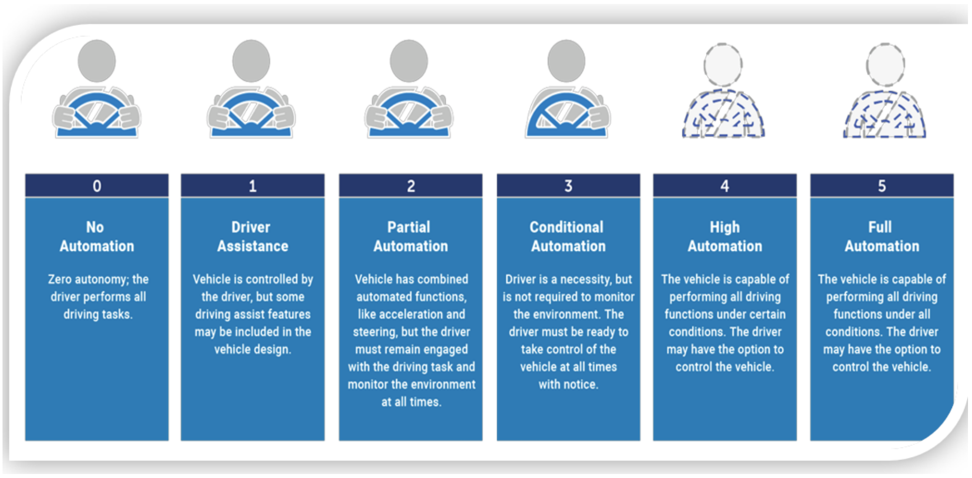
The quest for automation has relentlessly pushed the boundaries of technological innovation, aiming to render tasks more efficient, safer, and less reliant on human intervention. A striking exemplification of this pursuit is the realm of self-driving cars—a paradigm shift promising to revolutionize how we envision and experience transportation. Combining intricate algorithms, and the power of machine learning, autonomous vehicles hold the potential to streamline urban traffic, and adapt seamlessly to changing road scenarios using real-time image processing. This can significantly improve vehicular safety and efficiency, setting new benchmarks for the industry.

This study thoroughly examines the intersection of computer vision and machine learning and how they can be utilized to model and simulate self-driving cars. As self-driving cars rely heavily on interpreting their surroundings, computer vision acts as their cognitive system, enabling them to perceive and understand the environment. Coupled with machine learning algorithms, these vehicles perceive, learn, adapt, and make driving decisions analogous to human drivers with potentially reduced error rates.

The following chapters navigate through various facets of this captivating intersection of technology. Beginning with a comprehensive overview of the current feasible applications and advancements in simulator technologies, the research unravels both non-AI and AI-centric approaches to autonomous driving. This sets the foundation for a deeper dive into fundamental machine learning concepts, eventually leading up to the crux of the research—the design, implementation, integration, and testing of an autonomous driving model using Convolutional Neural Networks (CNNs). Alongside, Python, renowned for its vast applicability in data science and artificial intelligence (AI), is the foundational programming language driving the core of this research. Further enhancing our modeling capabilities is Keras, an efficient and high-level neural networks API, which serves as the primary conduit for constructing and refining our experimental models.

1.1 Background and Related Work

The concept of autonomous driving revolves around various 'autonomy levels,' each defining a distinct degree to which a vehicle can function without human intervention. The spectrum extends from cars equipped with essential driver assistance features to the envisioned fully autonomous vehicles. To truly understand how autonomous driving has evolved, we must dissect these autonomy levels outlined by the National Highway and Traffic Safety Administration (NHTSA) Figure 1.1 [1].



Zero autonomy; the driver performs all driving tasks.

Vehicle is controlled by the driver, but some driving assist features may be included in the vehicle design.

Vehicle has combined automated functions, like acceleration and steering, but the driver must remain engaged with the driving task and monitor the environment at all times.

Driver is a necessity, but is not required to monitor the environment. The driver must be ready to take control of the vehicle at all times with notice.

The vehicle is capable of performing all driving functions under certain conditions. The driver may have the option to control the vehicle.

The vehicle is capable of performing all driving functions under all conditions. The driver may have the option to control the vehicle.

Figure 1.1: Autonomous Driving Levels [1]

Levels of Autonomous Driving:

1. ***Level-0:*** No automation - In this stage, the human driver solely manages all driving operations without assistance from the vehicle's systems. Classic or vintage cars without modern assistance technologies would fall under this category [1].

2. ***Level-1:*** Driver assistance - At this level, the vehicle can assist with either steering or braking, but not both simultaneously. Features like Adaptive cruise control systems and lane-keeping assist systems are standard features for this category [8].

3. ***Level-2:*** Partial automation - Vehicles can simultaneously manage steering and acceleration/deceleration at this level. Despite this, the human driver should monitor the traveling environment while performing other duties. The Tesla Autopilot system is a well-known example of Level-2 automation, providing steering, braking, and acceleration assistance to the driver [9]

4. ***Level-3:*** Conditional automation - The vehicle under specific conditions (e.g., on highways) can control all driving tasks. However, the human driver must be ready to take control [8]. Audi's Traffic Jam Pilot,

capable of handling the vehicle in heavy traffic up to speeds of 60 km/h, exemplifies Level-3 automation [10].

5. ***Level-4:*** High automation - Vehicles operating at this level can assume complete control of all driving tasks under certain conditions without requiring human intervention or attention. If a change of affairs arises, the vehicle can halt and park itself securely. This level is exemplified by Google's Waymo's testing of Level-4 vehicles on public roads in 2017 [11].

6. ***Level 5:*** Full automation - At this ultimate level, the driving system takes complete control over the entire driving task under all circumstances, and the human driver does not need to be inside the car. While many autonomous vehicle developers aspire to this level, no authentic Level-5 vehicles exist yet [8].

As autonomy increases with each level, the imperative for solid and reliable systems to navigate complex real-world scenarios grows significantly, underscoring the need for rigorous testing and validation through some car-driving simulations to replicate the real world [1]. In the next chapter will discuss some of the most feasible applications of self-driving car simulations.

Chapter 2

Feasible applications of self-driving Cars

The evolution of driving simulators has mirrored and supported the progression of autonomous vehicle technology. With the emergence of more sophisticated machine learning and computer vision techniques in the 21st century, driving simulators began employing higher-fidelity models of road networks, vehicle dynamics, and traffic behavior [12]. The availability of open-source driving simulators like CARLA and AirSim among others, which even integrates weather patterns and pedestrian behavior, exemplifies the current state of simulator technology [13].
Additionally, the rise of Hardware-in-the-Loop (HIL) and Software-in-the-Loop (SIL) simulation methods allows for simultaneous testing of software and hardware components of autonomous vehicles in a controlled setting environment. [14]. In this section, we will cover some of the most common automated vehicle simulation platforms used in the industry today.

2.1 CARLA

CARLA (Car Learning to Act) is a noteworthy open-source driving simulator developed collaboratively by Intel Labs, the Toyota Research Institute, and the Computer Vision Center in Barcelona, Spain [5]. Designed to support the testing and development of autonomous driving (AD) technologies, CARLA strives to provide a realistic virtual environment that can mimic diverse real-world driving conditions [13].

CARLA's simulation framework incorporates various elements, including various vehicle models, urban layouts, pedestrians, buildings, and street signs. It allows for the flexible setup of sensor groups, delivering critical signals and data that can aid in training automated driving strategies. It can provide detailed data on metrics such as speed, acceleration, GPS coordinates, collisions, and other violations. The software's ability to generate different environmental conditions, such as time of day and weather, further augments its usefulness for autonomous vehicle testing. Importantly, it provides comprehensive technical feedback, which is crucial for the learning process of automated driving systems [13].

CARLA employs three different testing methodologies to study automated driving performance. The first involves a classic Modular Pipeline (MP) composed of a vision-based perception module, a rule-based planner, and a maneuver controller. The second and third methods utilize deep neural networks. One employs Imitation Learning (IL) to map sensor inputs to driving commands, while the other uses reinforcement learning (RL) to train autonomous driving agents through trial and error. These testing methodologies are employed in conjunction with a range of controlled, goal-directed navigation scenarios of increasing complexity [13].

As a result, none of the methods achieved perfect performance, indicating that the challenges of AD are far from fully solved. While the MP and IL methods showed comparable success, their performance was still less than ideal in the more complex tasks and scenarios. Despite the potential the RL has shown in other applications [15], it fell short in this study, demonstrating the brittleness and complexity of this approach when applied to autonomous driving. Therefore, further advances in learning algorithms, model architectures, and training diversity are needed to improve the performance and robustness of AD systems significantly [13].

2.2 Airsim

AirSim's significance as a testing platform for autonomous vehicles can be understood by the variety of research it has facilitated. Not only does AirSim support the development and validation of autonomous driving algorithms, but it also plays a crucial role in exploring novel techniques to bridge the reality gap – the difference between the simulation and real-world performance. It provides rich 3D visuals, and it's designed for hardware-in-the-loop with physically and visually realistic simulations [16].

AirSim is characterized by its high-fidelity visuals, physics, and sensor models. This enables autonomous vehicles' algorithms to be trained in a highly realistic virtual environment, which improves the transferability of these algorithms to the real world. The simulator allows for a variety of sensors, including LiDAR, GPS, and depth cameras. AirSim also offers a comprehensive API for researchers to manipulate the environment and vehicle dynamics [16].

It has been employed to train a Deep Deterministic Policy Gradient (DDPG) agent for autonomous driving. A study reported that combining reinforcement learning with the photorealistic simulation environment provided by AirSim resulted in better performance in lane-following and obstacle-avoidance tasks. Nevertheless, these studies also acknowledge that challenges remain. Despite AirSim's high-fidelity simulations, the reality gap still exists and poses difficulties in transferring the learned policies to real-world applications. Researchers highlight the importance of continued improvements in simulation realism and developing more effective Sim2Real transfer learning techniques. [17].

2.3 CarSim

CarSim is a high-fidelity, commercially available simulator extensively used in both academia and industry for developing, testing, and validating AD technologies. This software is known for its ability to accurately simulate the dynamic behavior of vehicles under a wide range of conditions and configurations. It's versatility can be attributed to its comprehensive vehicle models, road models, and detailed environmental settings. Its physics-based vehicle models, including passenger vehicles, commercial trucks, and trailers, consider various vehicle components and their dynamic interactions [18].

This simulator offers different road scenarios that cater to various driving conditions, including rural, urban, and highway driving. This range of scenarios enables a thorough examination of the vehicle's response under different circumstances. The simulator also supports incorporating traffic and pedestrians into the simulation, providing a more realistic evaluation environment [19].

CarSim also allows for integrating various sensor models, including camera, LIDAR, and RADAR, critical for developing and testing perception algorithms for autonomous vehicles. Additionally, CarSim has a Hardware-in-the-Loop (HIL) capability, enabling testing of actual hardware components in a simulated environment. This feature is particularly useful for testing sensor hardware and control units [19][20]. In addition to that, This approach is particularly promising for the evaluation and development of Advanced Driver-Assistance Systems (ADAS), enhancing safety and performance metrics in this evolving field [20].

However, future enhancements, including integrating more vehicle subsystems and control units in the HIL simulations for a more realistic representation of real-world conditions, are still required [20].

CarSim is proprietary software, making it less accessible for some researchers and developers compared to open-source alternatives. Nevertheless, it's advanced capabilities and high fidelity make it a favored choice for rigorous testing and validation of autonomous vehicle technologies [19] [20].

2.4 Udacity

Udacity, a pioneering platform in the field of online education, introduced a unique Self-Driving Car Engineer Nanodegree program aimed at nurturing future professionals in autonomous vehicle technology. While it isn't a simulator per se, as part of this comprehensive program, they have developed an open-source simulator that can be used to test self-driving algorithms [21].

This simulator is designed to support developing and testing autonomous driving algorithms in a virtual environment. It offers a variety of scenarios for testing different aspects of autonomous vehicle functionality, such as traffic light detection, steering control, and throttle control [21] [22].

The simulator's architecture includes a simple, user-friendly interface, primarily focusing on testing self-driving cars' functionality in different circumstances. It enables the users to gather data on vehicle's behavior under varying environmental and road conditions [22].

Udacity introduces the concept of end-to-end learning for control tasks, where a neural network learns to map raw sensor data directly to steering commands. This is demonstrated in Behavioral Cloning, where a deep learning model is trained to mimic human driving behavior based on data collected from simulator driving sessions [7].

These techniques are practiced within Udacity's simulator, providing a hands-on learning experience of applying deep learning to autonomous driving tasks. However, it's worth noting that while these methods have proven effective in the controlled environment of the simulator, translating them to real-world performance is a challenging task that requires rigorous testing and validation [7].

Chapter 3

Approaches to Autonomous Driving

The steering control mechanism ensures safe and accurate navigation in the autonomous driving industry. Various approaches have been employed to tackle the complex task of steering in self-driving vehicles. These approaches can be broadly categorized into three main categories:

- non-AI approach
- AI approach
- combination of AI and non-AI methods

Understanding these approaches is crucial for building effective and reliable steering mechanisms that enable autonomous vehicles to navigate diverse road scenarios with precision and safety.

3.1 Non-AI Approach

3.1.1 PID Control

PID Control (Proportional-Integral-Derivative) is a classic control strategy that adjusts the steering input based on three components [23]:

- *proportional* K_p- generates a control action proportional to the error between the desired and actual steering angles.
- *Integral* K_i- dampens oscillations by considering the rate of change of the error.
- *derivative* K_d- considers past errors to correct long-term biases or steady-state errors.

By tuning these components K_p, k_i, K_d, the PID controller minimizes the error and maintains stable and accurate steering control according to the mathematical formula below [23]:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)\, d\tau + K_d \cdot \frac{de(t)}{dt} \tag{3.1}$$

Where u(t) represents the control input (steering command) at time t and e(t) denotes the error between the desired set-point (desired steering angle) and the measured process variable (actual steering angle) at time t.

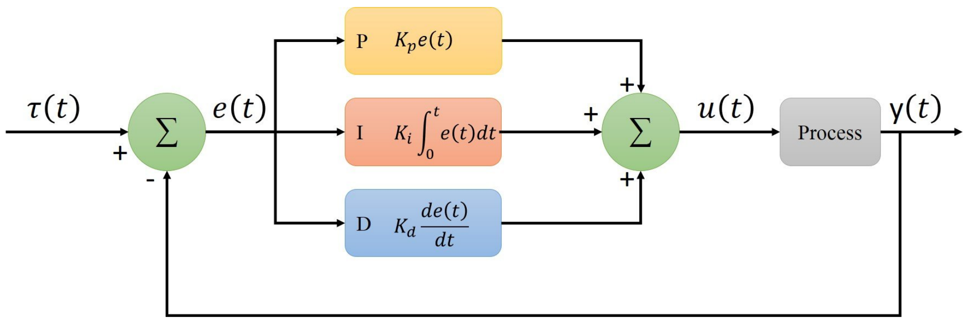


Figure 3.1: PID Block Diagram [2]

In a study examining autonomous vehicle simulation, a dual PID controller system was introduced the primary for lateral movement and the secondary for speed regulation. Though they operated independently, both controllers aimed for accurate trajectory following using discretized waypoints paired with velocity set-points. While safety and comfort were prioritized, and continuous tuning was applied, the lateral controller struggled with precise tracking on roads with pronounced curvatures. This suggests that while PID is adequate for lateral control, incorporating advanced methods like gain scheduling or superior controllers like MPC might achieve better outcomes [23]. Model Predictive Control (MPC) has shown efficacy in steering autonomous vehicles, successfully tracing various velocities and trajectories using a simple kinematic bicycle model [24]. While both MPC and PID controllers are commonly used in self-driving cars, MPC stands out due to its adaptability and ability to manage complex systems. Yet, it's computationally intensive, and while certain modifications can reduce this demand, they may compromise optimization. Therefore, learning-based or AI methods, which strike a balance between computational demands and performance, could be more suitable for real-world applications

3.2 AI-Approach

Artificial intelligence (AI) approaches in the context of autonomous vehicles involve utilizing advanced algorithms and techniques to enable vehicles to sense, perceive, understand, and make decisions autonomously [24].
The automotive AI industry will be dominated by deep learning technology, a method for applying Machine Learning (ML) (as part of being an AI subset as shown in figure 3.2, ML can be defined as a field that enables computers to learn and improve their performance without being explicitly programmed) [23]. It is now used in autonomous cars for speech recognition, voice search, sentiment analysis, recommendation engines, picture recognition, and motion detection [25].

Additionally, vehicles can perceive the environment by integrating computer vision with machine learning algorithms, therefore extracting meaningful insights and patterns from the visual data. This enables more sophisticated decision-making processes and enhances the vehicle's ability to navigate complex road scenarios [26].

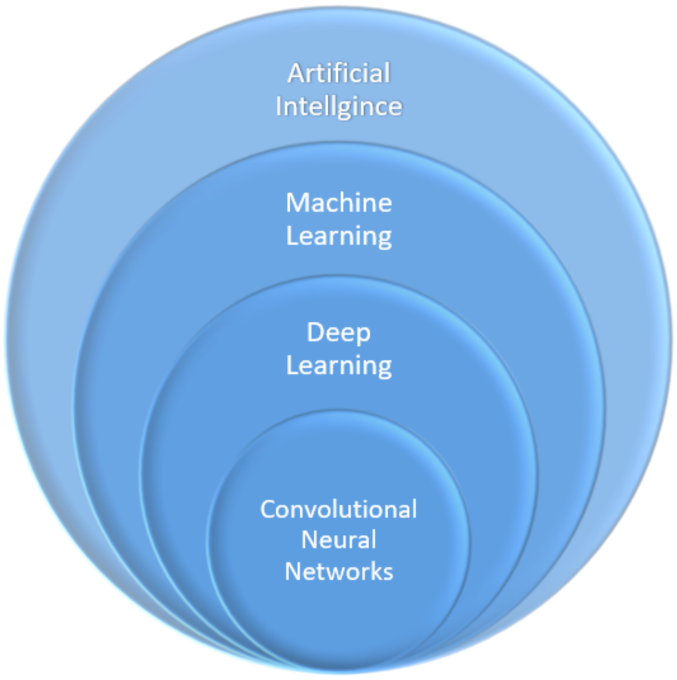


Figure 3.2: Relation between Artificial Intelligence, Machine Learning, and Deep-Learning

According to projections, the Automotive Artificial Intelligence market is set to expand from USD 2.3 Billion in 2022 to USD 7.0 Billion by 2027. This represents a Compound Annual Growth Rate (CAGR) of 24.1% over the five-year period [27]. The key factors contributed to this growth are:

- *Human-machine interface for infotainment systems* including voice and gesture detection, eye tracking, driver monitoring, virtual assistance, and natural language interfaces to recognize and classify the data that the driver uses to create a Level 3 autonomous system [28][29].
- *Autonomous cars and advanced driver assistance systems (ADAS)* including sensor-fusion engine control units (ECUs), radar-based detection units, driver condition assessment, and camera-based machine vision systems [30].

Chapter 4

Machine Learning

4.1 Supervised Learning

Supervised Machine Learning is a sub-field of machine learning where the algorithm learns from labeled training data that consists of input-output pairs. For example, the goal is to approximate a function f using a machine learning algorithm. In supervised learning, the function maps elements from an input domain $\mathbf{X}$ to elements in an output domain $\mathbf{Y}$. To approximate f, the machine learning algorithm is given a set of pairs of inputs and their corresponding ground truth outputs denoted $\mathcal{L} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where $f(x_i) = y_i$. The goal of the machine learning algorithm is to approximate f as close as possible, meaning the prediction $\hat{f}(x) = \hat{y}$ should be as close to the true value $f(x) = y$ as possible [31]. Supervised learning approaches are extensively

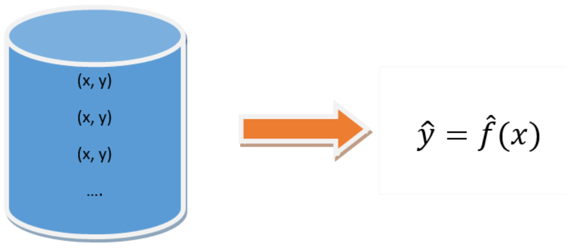


Figure 4.1: Supervised learning illustration

applied in computer vision techniques. Among these, methodologies rooted in deep learning, a subset of supervised learning, have consistently demonstrated superior outcomes in diverse tasks such as recognizing objects, classifying images, and performing semantic segmentation [32].
It is crucial to start with the differences between Linear and logistic regression, two widely used algorithms in machine learning through a supervised manner.

- *Linear regression:* used for predicting continuous numerical values [33].
- *logistic regression:* used for predicting the probability or likelihood of a binary outcome [34].

These algorithms have distinct equations and objectives, making it crucial to comprehend their differences to select the appropriate approach for end-to-end learning of a self-driving car.

4.1.1 Linear Regression

The training process involves presenting the algorithm with a set of input features and their corresponding known output labels, allowing the model to learn the underlying patterns and relationships between the inputs and outputs. Once trained, the model can make predictions or classifications on new, unseen data by generalizing from the learned patterns. The model takes the form of a linear regression equation of this type [35]:

$$y = w_0 + w_1 x \tag{4.1}$$

Where:

- y : Output/target variable
- x : Input/feature variable
- w_0 : Bias term or y-axis intercept
- w_1 : Regression coefficient or scale factor

The input variables are also known as independent, which signifies that they are not affected by other variables in the model and are assumed to impact the output variable directly [33]. We must consider the linear model and loss function to know the linear regression's representation.

Linear Model

In steering control, linear regression assumes a linear relationship between the input features (i.e., vehicle speed, heading angle, road conditions) and the predicted steering angle. For example, in the context of steering control, The linear model using a neural network can be expressed like this [33]:

$$\hat{y}_i = w_0 + w_1 \cdot \text{Speed} + w_2 \cdot \text{Heading Angle} + \ldots + \epsilon$$

The predicted steering angle $\hat{y}_i$ is the dependent variable, and the input features (speed, heading angle, etc.) are the independent variables. The coefficients $w_0, w_1, w_2, \ldots$ represent the impact of each feature on the steering angle, and ϵ is the error term, which tells us if our model is doing well or not.

Loss Function - Mean Squared Error

Linear regression aims to find the best-fit line that minimizes the difference between predicted and actual values, such as the mean squared error (MSE) that quantifies the discrepancy between the predicted and actual steering angles [36].

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.2}$$

where:

- n is the dataset's total number of data points.
- y_i represents the actual steering angle for the i-th data point.
- $\hat{y}_i$ represents the predicted steering angle for the i-th data point.

The linear model is used to make predictions, and the MSE is used to evaluate how good those predictions are or how well the model fits the data. The smaller the MSE, the better the fit, which means the model's predictions are closer to the actual values. During the training process, the objective is to minimize the MSE by adjusting the model's parameters. This is typically done using optimization algorithms such as gradient descent, which iteratively updates the parameters to minimize the MSE and improve the model's predictive performance [37].

4.1.2 Logistic Regression

In logistic regression, the objective is to predict the probability of an event or the likelihood of a binary outcome, typically represented as 0 or 1. Mathematically can be expressed as [34]:

$$\ln\left(\frac{y}{1-y}\right) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_kx_k \tag{4.3}$$

This is achieved using a threshold value, which determines the probability of an outcome being classified as 1 or 0. To model this relationship, logistic regression utilizes a Sigmoid function $\sigma(x)$ taking S-shape as shown in figure 5.2, also known as the logistic function [34].

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.4}$$

The sigmoid function transforms any real-valued input into a range between 0 and 1, suitable for representing probabilities [38]. However, accurately measuring these probabilities requires calculating the difference between them and the actual labels. The cross-entropy loss function is a useful tool for accomplishing this task.

Cross Entropy

Cross entropy is a mathematical function commonly used as a loss function in logistic regression and other classification models. It measures the dissimilarity between the predicted probabilities and the actual labels, measuring how well the model fits the observed data [39]. The cross-entropy function can be expressed mathematically as [40] [41]:

$$C = L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \tag{4.5}$$

where y is the true label, and $\hat{y}$ is the predicted probability. Understanding the mathematical foundations of linear and logistic regression provides a solid basis for developing accurate and reliable steering control systems. By utilizing linear/logistic regression techniques, autonomous vehicles can effectively learn and predict the appropriate steering angles based on various input features, enabling them to navigate the road with precision and safety [42].

Gradient Descent

Gradient descent is a fundamental optimization algorithm that is used to find the optimal parameters (weights) for either linear or logistic regression models. The general update rule for the weights in linear regression using gradient descent is given by iterations like this [43]:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{4.6}$$

Where:

- θ_j: the j-th weight parameter.
- α: learning rate, which controls how large the steps are during the descent.
- m: number of training samples.
- $h_\theta\left(x^{(i)}\right)$: hypothesis function representing the predicted value for the i-th training sample. For example:
  - For linear regression, $h_\theta\left(x^{(i)}\right)$ would be a linear combination of the features [44]:

    $$h_\theta\left(x^{(i)}\right) = \theta_0 + \theta_1 x_1^{(i)} + \ldots + \theta_n x_n^{(i)}$$

  - For Logistic regression, $h_\theta\left(x^{(i)}\right)$ is the sigmoid of the linear combination of the features:

    $$h_\theta\left(x^{(i)}\right) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1^{(i)} + \ldots + \theta_n x_n^{(i)})}} \tag{4.7}$$

- $y^{(i)}$: actual output for the i-th training example.
- $x_j^{(i)}$: the j-th feature value for the i-th training example.

This iterative process continues until convergence is achieved, where the model reaches a state of minimal loss and optimal parameter values. More Types of gradient descent will be discussed in chapter 6.2.3.

Chapter 5
Perceptron

The Perceptron, introduced by Rosenblatt in 1957, is a binary classification algorithm that mimics the functioning of a biological neuron. Acting as a single-layer feed-forward neural network, the Perceptron's significance extends to being the foundation of modern neural networks [45].
The basic operation of a perceptron involves summing weighted inputs and a bias, then applying an activation function (like the Step function as shown in 5.1), and ultimately making a binary classification decision. While powerful for linearly separable problems, its limitations and inability to solve problems like XOR (a non-linearly separable function), led to the development of Multi-Layer Perceptron (MLP). This evolution marked the transition from simple to complex neural models capable of handling non-linear patterns [46] [47].

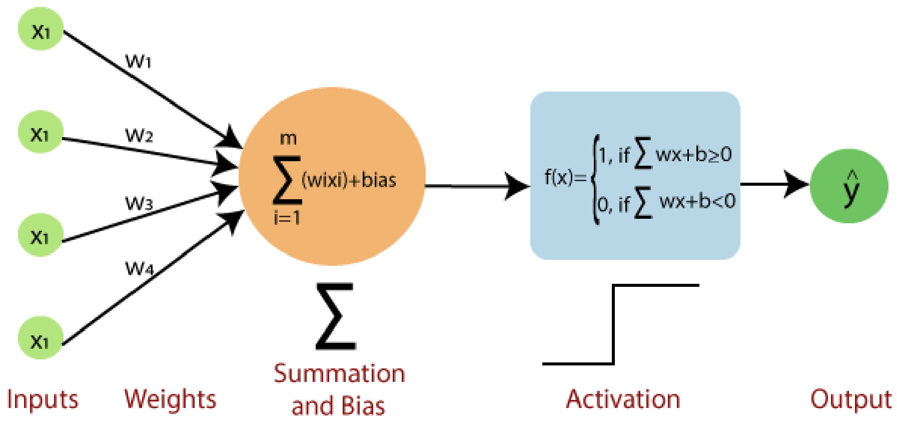


Figure 5.1: Perceptron Model ([3])

5.1 Mathematical Formulation of a Perceptron

The perceptron is a binary classifier that maps its input $\mathbf{x}$ (a real-valued vector) to an output value $f(\mathbf{x})$ (a single binary value) by passing it through a linear predictor function combined with a thresholding activation function. Formally, given an input vector $\mathbf{x}$ and a weight vector $\mathbf{w}$, the perceptron's output $f(\mathbf{x})$ is defined as [46]:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Where:

- $\mathbf{w}$: a vector of real-valued weights.
- $\mathbf{x}$: the input vector.
- b is the bias.
- $\mathbf{w} \cdot \mathbf{x}$ denotes the dot product of the vectors $\mathbf{w}.\mathbf{x} = \sum_{i=1}^{n} w_i x_i$.

The perceptron algorithm learns the appropriate weight coefficients from the training data so that the instances of one class are separated from the instances of the other class by a hyperplane.

5.1.1 Activation Functions

The activation function in the perceptron model in figure 5.1 plays a crucial role in determining the model's output. Specifically, the perceptron uses a step function as its activation function, introducing a thresholding non-linearity. This allows the perceptron to make binary decisions by separating the input space with a hyperplane.

However, this particular non-linearity is quite simple, and it can limit the model's ability to learn more complex patterns. In the context of more advanced neural network models, different activation functions, such as sigmoid or ReLU, are used to introduce more flexible non-linearities, enabling the network to capture more complex relationships in the data. Table 5.1 summarizes the frequently used activation functions in Deep Neural Networks (DNN) with their graphical representation in figure 5.2 [48] [49].

Activation Function	**Equation and Description**
Step function	$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$ Here $z = w^T \cdot x + b$, the dot product of the weight and input vectors plus the bias term, is the input to the activation function.
Sigmoid function	$f(x) = \frac{1}{1+e^{-x}}$ The sigmoid function squashes real numbers between 0 and 1.
Hyperbolic Tangent (Tanh) function	$f(x) = \frac{2}{1+e^{-2x}} - 1$ The tanh function squashes real numbers in the range of -1 to 1.
ReLU (Rectified Linear Unit) function	$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$ ReLU outputs the input directly if it's positive; otherwise, it outputs zero.
ELU (Exponential Linear Units) function	$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$ ELU tends to converge cost to zero faster and produces more accurate results.
Softmax function	$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$ The softmax function calculates the probabilities of each class in a multiclass classification problem.

Table 5.1: Common Activation Functions

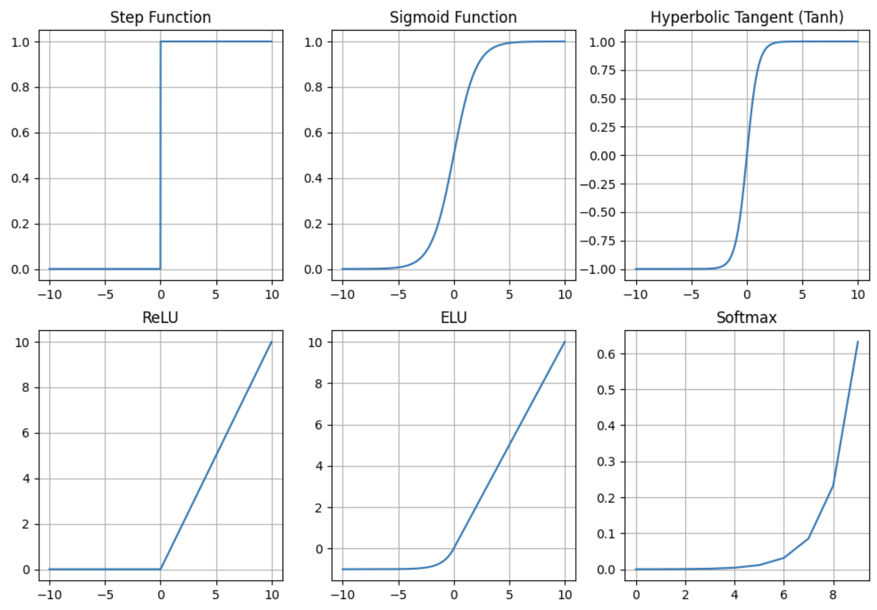


Figure 5.2: Common Activation Functions

Chapter 6

Deep Neural Network (DNN)

Deep Neural Networks (DNNs) extend the single-layer perceptron by introducing the concept of ***hidden layers***. This added complexity enables the modeling of nonlinear and hierarchical features. Marking DNNs a versatile tool in numerous applications such as image recognition [50], speech processing [51], and natural language processing [52].

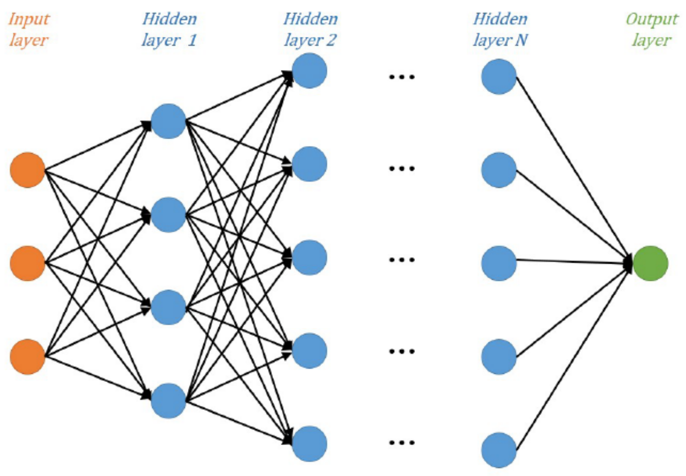


Figure 6.1: DNN Layers [4]

Deep Neural Networks comprise multi-layers of perceptron, each serving a distinct purpose in the overall network. It starts with the input, hidden, and output layer [53]:

- *Input Layer:* accepts the raw input data, each neuron representing an individual feature in the data set.
- *Hidden Layers:* perform transformations on the inputs received from the preceding layers. The "depth" of a network refers to its number of hidden layers.
- *Output Layer:* provides the network's final output, which is the prediction or classification made by the network.

6.1 Feed-forward Neural Network

Each layer in a DNN typically consists of multiple neurons (or nodes) as shown in figure 6.1. These neurons perform linear transformations on their inputs, followed by a non-linear activation function. The output of one layer serves as the input for the next layer. This hierarchical structure allows DNNs to model complex patterns in the input data by combining simpler patterns learned in the lower/previous layers [53].
Mathematically, the computation performed by a layer in a DNN is almost similar to the perceptron in equation 5.1. Considering MLP, the equation can be represented in the form below [54]:

$$a = \phi\left(\sum_j w_j x_j + b\right) \tag{6.1}$$

Where:

- a: Activation/output of the layer.
- ϕ: Activation function applied to the weighted sum of inputs.
- w_j: Weights applied to the inputs.
- x_j: Inputs to the layer.
- b: Bias term.

The equation describes the computation performed by a neuron/layer in a DNN. The weighted sum of the inputs (i.e., $\sum_j w_j x_j$) is first computed, and the bias b is added. This total is then passed through the activation function ϕ, resulting in the activation a of the neuron. The process is repeated across all neurons and layers in the network to compute the final output y.
To express y using a ***fully connected network***, we must include the calculations for all the layers that lead up to the output layer. For example, assuming DNN with L layers, activation function ϕ_l of layer l, weights W_l, and the biases b_l, we can represent the computations as [54]:

$$\begin{aligned} a^{(1)} &= \phi_1(W_1 x + b_1) && (6.2)\\ a^{(2)} &= \phi_2(W_2 a^{(1)} + b_2) && (6.3)\\ &\vdots && (6.4)\\ a^{(L-1)} &= \phi_{L-1}(W_{L-1} a^{(L-2)} + b_{L-1}) && (6.5)\\ y = a^{(L)} &= \phi_L(W_L a^{(L-1)} + b_L) && (6.6) \end{aligned}$$

Here:

- x is the input vector.
- W_l is the weight matrix for layer l.

- b_l is the bias vector for layer l.
- ϕ_l is the activation function for layer l.
- $a^{(l)}$ is the activation vector for layer l.
- $y = a^{(L)}$ is the final output of the network.

These equations describe a feed-forward pass through the network, where the output of each layer serves as the input to the next layer, culminating in the final output y.
After applying activation functions to each layer in the feedforward process of a DNN, the network has completed its computation of the input response. The subsequent step typically involves a loss or objective function that measures how well the network's output matches the desired target. This process is essential for training the network [54].

6.2 Training Deep Neural Network

Once the activation functions have been applied to each layer in the feed-forward process of a DNN and the network has fully computed its response to the input. The next step usually involves a loss function that quantifies how well the network's output aligns with the expected target. This is crucial for training the network. Furthermore, an essential mechanism for proper training of the network is backpropagation, as it utilizes the chain rule to compute gradients, facilitating weight updates to minimize the loss function.

6.2.1 Loss Function

A loss (also called cost or objective) function is used to measure the precision of the network outcome. This function determines how much the prediction deviates from the anticipated result. The result of this function is a numerical value, also known as the penalty or cost. For instance, the cross-entropy loss, MSE functions explained in chapter 4.1.2 is frequently used in visual classification tasks where probabilities are computed [53].

6.2.2 Back-propagation

For supervised learning, target classes are essential for error calculation. The error is afterward backpropagated to every node in previous layers. This error is obtained as a gradient of the cost/loss function C with respect to each layer's weights w_{ij}, given input of the node x and activation function ϕ. The method for calculating the error term differs depending on whether it is for the output layer or a hidden layer [41].

Output Layer Error:

For the output layer (L), the error term is often computed as [41]:

$$\delta^L = \nabla_a C \odot \phi'(z^L) \tag{6.7}$$

Here, $\nabla_a C$ is the gradient of the cost function C with respect to the activation neuron ($\nabla_a C = \frac{\partial C}{\partial a_j^{[L]}}$), ϕ' is the derivative of the activation function, $\odot$ - Hadamard product denotes element-wise multiplication, and $z^{[l]}$ is the weighted input vector for layer l, computed as $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$.

Hidden Layer Error:

For the hidden layers, the error term can be computed using the error term of the next layer [41]:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{6.8}$$

Here, w^{l+1} is the weight matrix for layer $l+1$, δ^{l+1} is the error term for layer $l+1$, and T denotes the transpose.

The rate of change of the cost with respect to the weights and biases in the network is vital for the parameter update process. These gradients are given by :

$$\frac{\partial C}{\partial w_{ij}^l} = a_j^{l-1} \cdot \delta_i^l \tag{6.9}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{6.10}$$

Gradient computation demands the application of the chain rule in order to compute partial derivatives of the loss function C with respect to particular weight w_{ij}. Using this error, weights are updated by an optimization algorithm such as gradient descent [41].

6.2.3 Gradient Descent Types

During backpropagation, the weights and biases in the network are updated using gradient descent. Recalling the theory obtained from chapter 4.1.2. The main objective is to minimize the cost function C, by adjusting the weights w_{ij}^l and biases b_j^l in the direction that reduces the gradient of the cost function. The adjustments are controlled by the learning rate η, and the gradients $\frac{\partial C}{\partial w_{ij}^l}$ and $\frac{\partial C}{\partial b_j^l}$ are calculated during back-propagation from equations 6.9 and 6.10 . Then the update formulas for weights and biases can be formulated as [41]:

$$w_{ij}^l = w_{ij}^l - \eta \frac{\partial C}{\partial w_{ij}^l} \tag{6.11}$$

$$b_j^l = b_j^l - \eta \frac{\partial C}{\partial b_j^l} \tag{6.12}$$

Gradient descent comes in three variants Batch Gradient Descent, Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent [55]. A small comparison of these algorithms, illustrating their updated equations, is presented in Table 6.1, where θ is the parameters of the model being optimized [55].

Algorithm	**Equation**
Batch Gradient Descent	$\theta = \theta - \eta \cdot \nabla J(\theta, X, y)$
Stochastic Gradient Descent (SGD)	$\theta = \theta - \eta \cdot \nabla J(\theta, x^i, y^i)$
Mini-Batch Gradient Descent	$\theta = \theta - \eta \cdot \nabla J(\theta, x^i : i + n, y^i : i + n)$

Table 6.1: Comparison of Gradient Descent Optimization Algorithms

The differences between these variants primarily revolve around the number of training examples used to compute the gradient at each step. In Batch gradient descent, which computes the gradient of the loss function using the entire training set X before updating the model parameters. However, this method is usually slow and requires the full training set to be kept in memory, which can be challenging for deep learning. The second variant SGD updates the weights after each training sample x_i. Although this method is faster, it can produce a lot of fluctuation. On the other hand, The most commonly used type is mini-batch gradient descent, which computes the gradient for a batch of training samples $x_{i:i+n}$. This method strikes a balance between computational efficiency and stability in the learning process, as it only requires a smaller subset of the training data to be in memory at any given time.
Furthermore, Finding the optimal learning rate η in DNN is tricky. It needs to be fast enough for efficient convergence but not so fast that it causes fluctuation. Gradient descent can lead to sub-optimal states, but optimizers like RMSprop and Adam address this issue [55]. For example, RMSprop works well with RNNs, while Adam performs better with CNNs and is generally faster than traditional gradient descent [56].

6.3 Types of Deep Neural Networks

Having explored the fundamental building blocks of training Deep Neural Networks (DNNs), from the definition of layers, and loss function to the details of back-propagation and gradient descent algorithms, we are now equipped with the essential understanding of how DNNs learn.

Neural Networks (NNs) come in different types, such as Perceptrons, Shallow Neural Networks (SNNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). Although they share the same fundamental concept of interconnected nodes or "neurons," their architectures and mechanisms differ. For example, Perceptrons and SNNs typically have only one or a few layers, while DNNs have multiple hidden layers. CNNs are ideal for image processing because they use convolutional layers to capture spatial hierarchies. RNNs have internal memory to handle sequential data. Despite these differences, all NNs types rely on weighted connections, activation functions, and the ability to learn from data. A comparison between types of neural networks and their application represented in table 6.2

Type	Layers	Key Features	Typical Applications
Perceptrons	Single layer	Simplest form of neural network, linear classifier	Binary classification
SNNs	One or few	Simplicity, limited hidden layers	Simple pattern recognition
DNNs	Multiple hidden	Multiple hidden layers, complex representations	Image, text, and speech analysis
CNNs	Multiple	Convolutional layers, pooling, spatial hierarchies	Image and video processing
RNNs	Multiple	Sequential processing, internal memory	Time series, speech recognition

Table 6.2: Comparison of different types of neural networks

Recurrent Neural Network - RNN

Recurrent neural networks are a class of deep neural networks that form connections between units over time, enabling them to process sequential data and maintain information about previous steps in a sequence [57]. The first development in RNNs was in the 1980s by John Hopfield [58]. In this network, the state of the nodes is activated depending on the input it receives from every other node. In 1989, the Elman network introduced the idea of inputs from previous time steps fed forwards ((passing previous outputs as inputs to the next layer)) [59]. Since then, RNNs have been demonstrated to be successful in speech recognition [57], natural language processing [60], and time-series prediction [61].

The basic algorithm of RNN is similar to a feed-forward neural network layer but includes a separate set of weights to evaluate the results of previous time step elements. The following equation can describe this [57]:

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b) \tag{6.13}$$

$$y_t = W_{hy}h_t + b_y \tag{6.14}$$

Where f is an activation function, h_t represents the current hidden state, h_{t-1} is the previous hidden state, x_t is the current input, W_{hh} and W_{xh} are the weight matrices, and b is the bias.

While this architecture allows RNNs to model sequential data effectively, it also makes them susceptible to challenges such as vanishing and exploding gradients, especially when dealing with long sequences. Various modifications, such as Long Short-Term Memory (LSTM - gradient-based method [62]) units, proposed to mitigate these issues [57].

Convolution Neural Networks - CNN

In 2010, the ImageNet Project exhibited the full potential of Deep CNNs. It has held an annual competition challenging teams to classify a large visual database with over 14 million annotated images [63]. While in 2010 a reasonable error rate was considered to be 25% [63], a deep CNN achieved an error rate below 5% in 2017 [64]. This significant advancement attracted both industry and research attention, leading to a surge in deep neural networks. This surge inspired NVIDIA to explore the use of deep CNNs in self-driving cars [7].

CNNs specifically designed to process data with grid-like topology, such as images organized in a 2D or 3D grid of pixels, making them a vital tool for computer vision tasks [53]. Inspired by biological visual perception, CNNs were first introduced by Fukushima in the 1980s [65]. They comprise multiple convolutional layers that apply filters to the input data, capturing local features and preserving spatial relationships [66]. Since their inception, CNNs have been a cornerstone in various applications such as image classification [50], vehicles & lane detection [67], and even medical imaging [68].
The basic architecture of a CNN is represented in figure 7.1 consists of a sequence of layers. This combination of layer types enables CNNs to automatically and adaptively learn spatial hierarchies from the input data. The next chapter will delve deeper into the architectures, mechanisms, and innovative applications of CNNs to apply it in our simulator to build a self-driving car being driven autonomously.

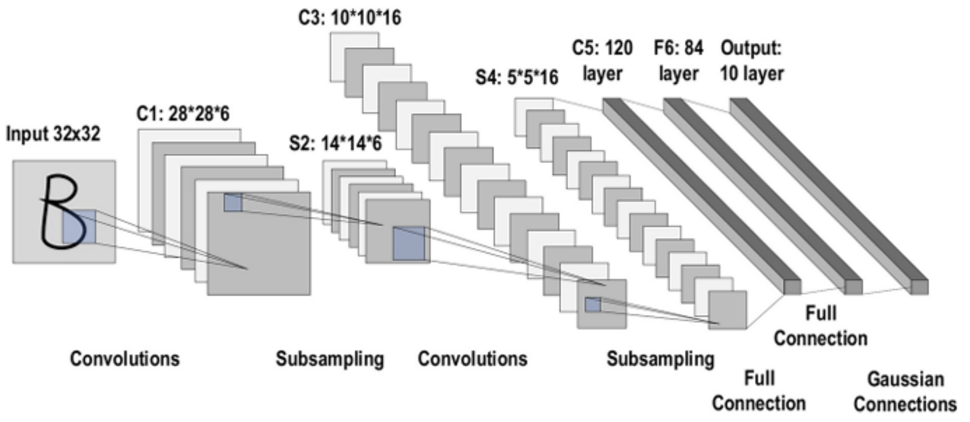
Chapter 7

Convolutional Neural Network (CNN)

CNN is commonly used in visual feature extraction tasks since they have the ability to capture edges or common characteristics in the image. For example, a CNN can be trained to recognize signs, detect pedestrians, and analyze traffic patterns [69]. Additionally, CNN analyzes visual data captured by the vehicle's cameras to enable the self-driving system to make real-time decisions and safely navigate through the environment. [7].

7.1 CNN Architecture

Convolutional Neural Networks consist of a series of layers designed to automatically and adaptively learn spatial hierarchies from data. The architecture of a typical CNN is composed of several types of layers as illustrated in figure 7.1: Convolutional layers, Pooling (subsampling) layers, Flatten layer, and Fully connected layers.



LeNet Architecture [66]

Figure 7.1: CNN Architecture

CNN takes image pixels as input. The information of an image can be represented as a 2D array for grayscaled images or a 3D matrix for RGB images with dimensions of (H x W x C), where H represents height, W represents width, and C represents channel (or depth). Each value of this matrix can be a color value in [0, 255] or a scaled version in [0, 1] for the corresponding color channel. A computer then processes this image data through a process called image processing [6].

7.1.1 Convolution Operation

The following equation can describe a typical convolutional operation [53]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{7.1}$$

Where:

- $S(i,j)$: the output of the convolution operation at spatial location (i,j).
- I: The input image.
- K: The 3D kernel or filter.
- $\sum_m \sum_n \sum_c$: Denotes a summation over m, n, and c. m and n define the kernel size, and c is the channel depth.

The convolution operation involves sliding a filter or kernel across the input image (or input feature map) and performing element-wise multiplication between the kernel values and the corresponding values in the input. This is then summed to produce a single value in the output feature map. The process is repeated for every location the filter can reach on the input, thereby producing the entire output feature map [6]. This operation is visualized in figure 7.2.

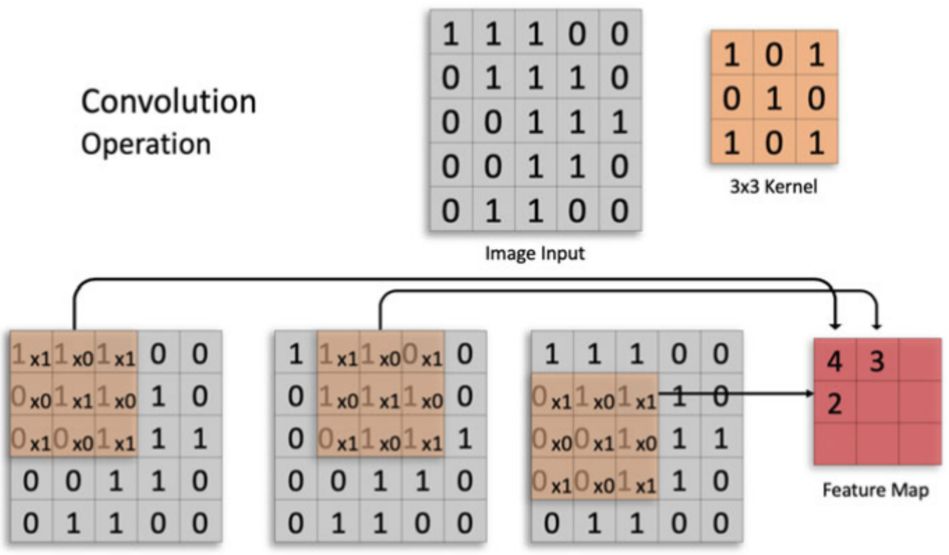


Figure 7.2: Convolution Operation with *3x3 filter and stride = 1* [5]

7.1.2 Pooling

This layer sub-samples feature maps to reduce variance within local regions of the image [6]. It splits the image into rectangular regions and takes out the value determined by the type of pooling layer. The most popular type of pooling layer in CNNs is the max-pooling layer, which extracts the maximum value of the sub-regions of the feature map, other pooling functions can be shown in figure 7.3.

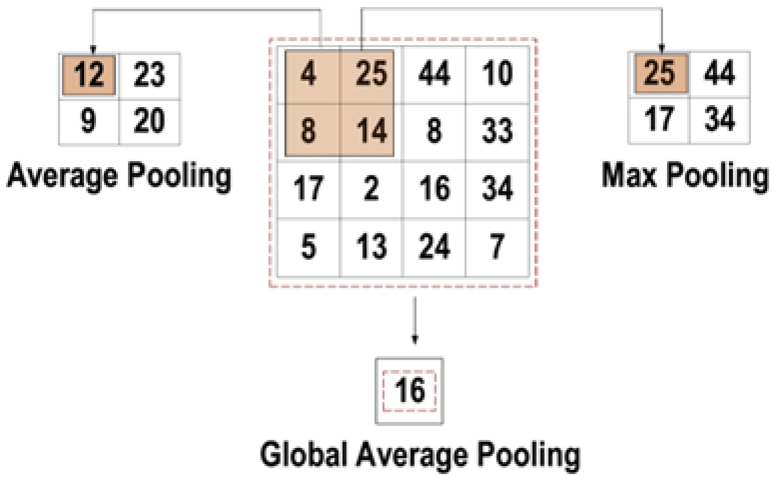


Figure 7.3: Pooling Operation with *2x2 filter and stride = 2* [6]

In the pooling operation, no parameters are learned; rather, it's a fixed operation designed to expedite the computation and promote feature invariance. However, the choice of pooling window size, stride, and type of pooling (max or average) can be considered hyper-parameters of the model and adjusted during the model tuning process [6] [53].

7.1.3 Fully Connected Layer

Towards the end of the network, there are one or more fully connected layers, also known as *dense layers*. These layers function like those in a standard MLP network. Their purpose is to perform high-level reasoning and make the final decision based on the features extracted by the convolutional layers.

Before passing the output from the last pooling or convolutional layer to the fully connected layer, the data is *flattened* (i.e., transformed from a 2D matrix to a 1D vector). The last fully connected layer produces the output of the network. This layer typically uses the softmax activation function for classification tasks, which outputs a probability distribution over the classes.

Chapter 8

Design & Implementation of Model

8.1 Overview

This project focuses on developing a machine-learning algorithm to predict a self-driving car's steering wheel angle. To achieve this, we collect and download relevant data, balance it, and divide it into training and validation sets. We then preprocess the images and establish a connection using Flask and Socket.io to create a bidirectional communication between the model and the Udacity simulator to test the car being driven autonomously without human intervention in real time.

End to End Self-Driving

End-to-End Self-Driving represents a pivotal advancement in autonomous driving, where neural networks directly process inputs like camera streams and vehicle parameters to perform complex tasks such as object detection and pathfinding [7]. The final output consists of a controlled direction for steering, accelerating, and braking of the car. The pioneering work by NVIDIA, which utilized a deep CNN to predict steering angles directly from raw pixels, has served as the foundation for this concept. The experimental section of this thesis aims to explore further and implement this integrated approach, reflecting the continuing evolution of end-to-end systems in the field of autonomous driving

Car Simulator - Udacity

This research used Udacity to create virtual training tracks and scenarios closely mimicking real-world conditions. This enabled the effective training and validation of the neural network, reinforcing the overall robustness of the autonomous driving system.

Keras

Keras is a high-level neural networks API written in Python and capable of running on top of TensorFlow [70]. It simplifies the process of building, training, and deploying neural networks, making it particularly useful for rapid prototyping and experimentation. In the context of this research, Keras was employed to construct and train the CNN responsible for predicting steering angles. Its streamlined interface and flexibility greatly facilitated the iterative design and optimization process, leading to more efficient and effective model development by offering layers such as Convolution2D, MaxPooling2D, Dropout, Flatten, and Dense. Additionally, the Keras API provides a variety of popular activation functions, including Relu among others as mentioned before in chapter 5.1.1

With these tools, the possibilities of self-driving techniques can be fully explored, from start to finish, in a unified and integrated development environment. The combination of Udacity's realistic simulation capabilities, Keras's user-friendly neural network interface, and Python's general-purpose programming flexibility provided a robust platform for the experimental design and validation of this research towards developing an autonomous driving model.

8.2 Data Collection

Manual Control and Data Documentation

To begin collecting data for training the end-to-end DNN model, we must manually operate the vehicle in the simulator by completing 2-3 training laps through the training mode. The data acquired during this process includes images that display the features and corresponding steering angle, speed, throttle, and reverse values as labels. This information is vital as it forms the features and labels that will be used to train a supervised machine-learning model. Additionally, it enables the neural network to understand the intricate relationships necessary for self-driving cars, ensuring that the model can interpret visual inputs and generate appropriate steering commands.

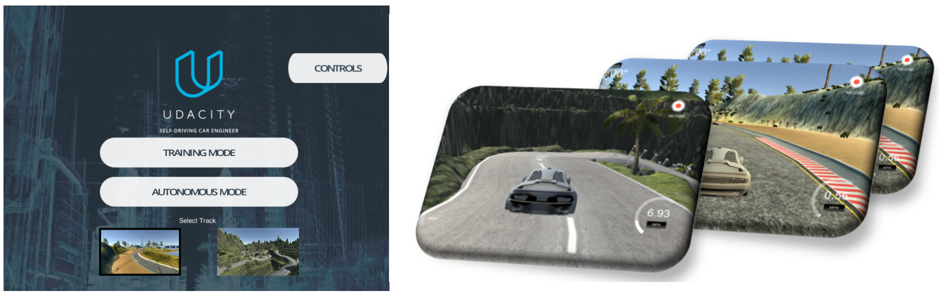


Figure 8.1: Udacity Self-Driving car environment

Data Preprocessing

Once the data has been collected, it must be organized and formatted to be suitable for training the neural network. This involves a series of steps executed through a Python script, leveraging the Pandas library for data manipulation.

The steering angle data is initially analyzed and represented in figure 8.2, revealing a significant bias value towards 0 angles, indicative of straight driving (as the car is usually driven in a straight line). Training the model based on the data provided creates a problem for the neural network, which could bias the model toward predicting a zero angle. Thus the car becomes biased towards driving straight all the time. To mitigate this issue, we rejected all the samples above a certain threshold of 200 samples per bin to ensure that the data is more uniform and not biased towards a specific steering angle.

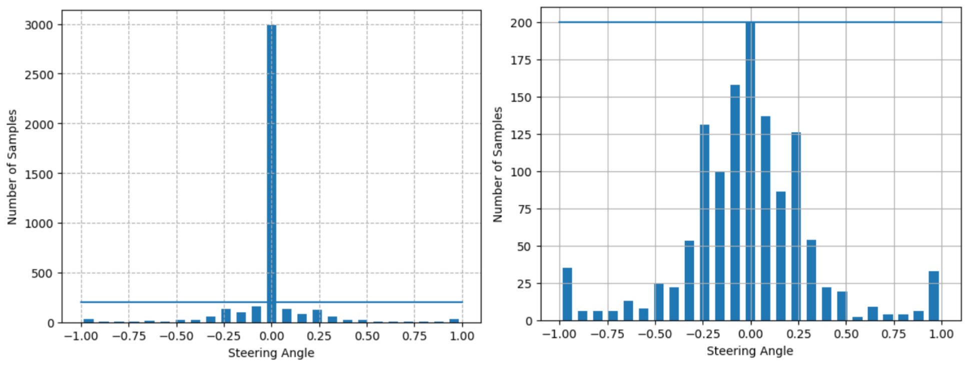


Figure 8.2: The histogram on the left visualizes the raw data of 4053 samples, after rejecting some of the biased values, there were only 1263 samples left for training

This preprocessing sequence prepares the data to be divided into training and validation sets, which will then be used to train the end-to-end self-driving car model using a deep neural network.

8.2.1 Training & Validation Split

A crucial phase in building a self-driving car utilizing computer vision and machine learning algorithms involves preparing and partitioning the data set into training and validation subsets. Image paths and corresponding steering angles were loaded into arrays, representing the steering controls for left, center, and right camera views. This is a standard practice in machine learning to evaluate the model's performance and to ensure that it can generalize well to unseen data.

```
X_train, X_valid, y_train, y_valid =
↪  train_test_split(image_paths, steering, test_size=0.2,
↪  random_state=6)
```

Figure 8.3: `train_test_split` function to split the data into training and validation subsets

Leveraging the `train_test_split` function from Scikit-learn, the data was randomly split into training (80%) and validation 20% (`test_size` = 0.2) sets.

—-	Training	Validation	Total
Number of samples	1010	253	1263
Percentage of total dataset	80%	20%	100%

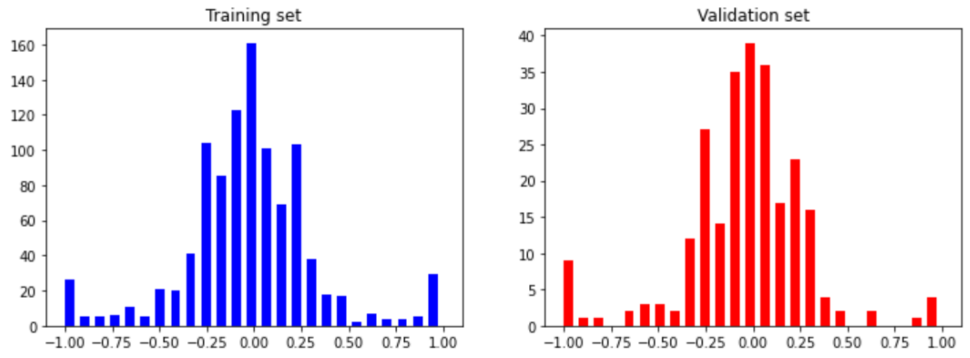


Figure 8.4: Training & Validation Set Histogram Representation

This ensured a well-balanced distribution of steering angles, aiding in model generalization. Histograms were utilized to visually confirm the consistent distribution between training and validation sets, facilitating an honest evaluation of the model's performance on unseen data as represented in figure 8.3.
From figure 8.3. It's clear that the data isn't exactly identical, but both follow a general trend such that both right and left steering angles between the training & validation sets are very similarly distributed, and the data is still biased towards 0 angle as required, since most of the time the car follows a straight line unless it needs to take a turn.

Subsequently, this collected dataset will be utilized later to train our DNN based on the training data to learn the appropriate steering angles according to which part of the track it's in extracting the necessary features in the image and then test it on the validation set to determine whether our neural model is under-fitting or over-fitting based on the steering angles predicted by the neural network in comparison to the actual steering angles.

8.2.2 Image Pre-processing

In the image preprocessing stage of the autonomous driving project, a sequence of critical transformations is applied to prepare the data for neural network training. this sequence is represented in the code snippet below.

```
def img_preprocess(img):
  img = mpimg.imread(img)
  img = img[60:135,:,:]
  img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
  img = cv2.GaussianBlur(img, (3,3), 0)
  img = cv2.resize(img, (200, 66))
  img = img/255
  return img

X_train = np.array(list(map(img_preprocess, X_train,)))
X_valid = np.array(list(map(img_preprocess, X_valid,)))
```

Figure 8.5: Code for preprocessing the image.

Initially, the images are cropped to extract our region of interest, thereby eliminating irrelevant features such as the sky & trees. The color space is then converted from RGB to YUV, which separates brightness from color, enhancing the model's sensitivity to structural patterns over mere color variations. Also, Gaussian blur is applied to remove high-frequency noise and accentuate significant structures. Then images are resized to a uniform dimension (200x66) to meet the neural network's input size requirement, followed by normalization of pixel values to fall within the range of 0 to 1.

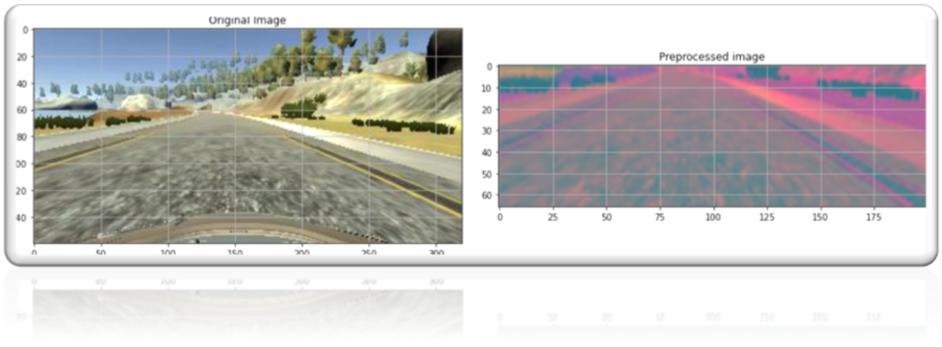


Figure 8.6: Original Vs Preprocessed Image

Lastly, we apply the necessary preprocessing to the entire training and validation datasets, preparing them to train and evaluate the neural network model for autonomous driving. This step ensures uniformity in the input data and optimizes the conditions for efficient and stable convergence during the training of the CNN, thus preparing the images to be fed into the machine learning algorithm for steering wheel angle prediction.

8.2.3 Model Selection

The project employed NVIDIA's end-to-end CNN as a reference, which takes input images in YUV space (66x200x3) to map pixels from a front-facing camera to steering commands [7]. This architecture represented in figure 8.9 consists of 5 convolutional layers and 4 fully connected/ Dense layers, with Exponential Linear Unit (ELU) activation function to introduce non-linearity as shown in the code snippet of the model in figure 8.7 below.

```
def nvidia_model():
  model = Sequential()
  model.add(Convolution2D(24, kernel_size=(5,5), strides=(2,2),
  ↪  input_shape=(66,200,3), activation='elu'))
  model.add(Convolution2D(36, kernel_size=(5,5), strides=(2,2),
  ↪  activation='elu'))
  model.add(Convolution2D(48, kernel_size=(5,5), strides=(2,2),
  ↪  activation='elu'))
  model.add(Convolution2D(64, kernel_size=(3,3),
  ↪  activation='elu'))
  model.add(Convolution2D(64, kernel_size=(3,3),
  ↪  activation='elu'))
  model.add(Dropout(0.5))
  model.add(Flatten())
  model.add(Dense(100, activation='elu'))
  model.add(Dropout(0.5))
  model.add(Dense(50, activation='elu'))
  model.add(Dropout(0.5))
  model.add(Dense(10, activation ='elu'))
  model.add(Dropout(0.5))
  model.add(Dense(1))
  optimizer = Adam(lr=1e-3)
  model.compile(loss='mse', optimizer=optimizer)
  return model
```

Figure 8.7: NVIDIA model Implementation.

The first 3 convolutional layers, using 5×5 kernel size and 2×2 stride, efficiently reduce spatial dimensions and emphasize key features. The next 2 layers employ *nonstrided* convolutions & 3×3 kernel size to retain spatial information, followed by a flattening process and 4 dense layers with dropout to avoid over-fitting. The final dense layer predicts the steering angle.

In addition, the model is compiled using Adam optimizer instead of Batch Gradient Descent since this intrinsically implements learning-rate decay as well as momentum and is very useful for better convergence and preventing overshooting [56]. Furthermore, since we are addressing a regression problem requiring precise continuous predictions of steering angles, the model was compiled using MSE as the loss function. As a result, the model has a total of 252,219 trained parameters, as shown in table 8.1. This encapsulates the model's complexity and its ability to extract intricate features from the input data.

8.2.4 Model Evaluation

In table 8.1, the sequential decrease in spatial dimensions through the convolutional layers, followed by the expansion and subsequent condensation in the dense layers, illustrates the model's process of distilling the vital information needed to make the steering predictions.

Layer (type)	Output Shape	Param #
`conv2d_5 (Conv2D)`	(None, 31, 98, 24)	1824
`conv2d_6 (Conv2D)`	(None, 14, 47, 36)	21636
`conv2d_7 (Conv2D)`	(None, 5, 22, 48)	43248
`conv2d_8 (Conv2D)`	(None, 3, 20, 64)	27712
`conv2d_9 (Conv2D)`	(None, 1, 18, 64)	36928
`dropout_2 (Dropout)`	(None, 1, 18, 64)	0
`flatten_1 (Flatten)`	(None, 1152)	0
`dense_4 (Dense)`	(None, 100)	115300
`dropout_3 (Dropout)`	(None, 100)	0
`dense_5 (Dense)`	(None, 50)	5050
`dropout_4 (Dropout)`	(None, 50)	0
`dense_6 (Dense)`	(None, 10)	510
`dropout_5 (Dropout)`	(None, 10)	0
`dense_7 (Dense)`	(None, 1)	11
Total params:	252,219	
Trainable params:	252,219	
Non-trainable params:	0	

Table 8.1: Summary of the Sequential Model

The graphical representation of training and validation losses over 30 epochs represented in figure 8.8 is an insightful gauge of the model's performance and the architecture's suitability for learning steering patterns in autonomous driving. Starting with a higher value, both losses decrease sharply as the network begins to learn, eventually slowing down and converging to a minimal and similar value. The *training loss* demonstrates the model's capability to fit the data, while the *validation loss* provides an unbiased evaluation of how well the model generalizes to unseen data. The resulting validation loss of **0.045** is not only a testament to the model's accuracy but also indicative of its robustness. This convergence of loss values is particularly significant, as it reflects a balance between learning the complex underlying patterns (avoiding underfitting) and not over-adapting to the training data (avoiding overfitting). Such a balanced model is essential for reliable and safe steering predictions, as it implies that the model is likely to perform well not only on the data it was trained on but also on new, unseen road conditions.

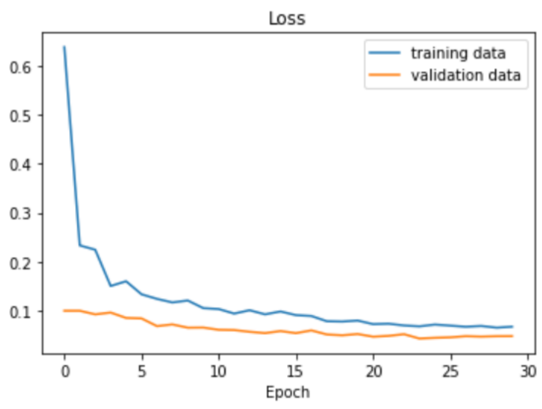


Figure 8.8: Validation Vs Training Data loss

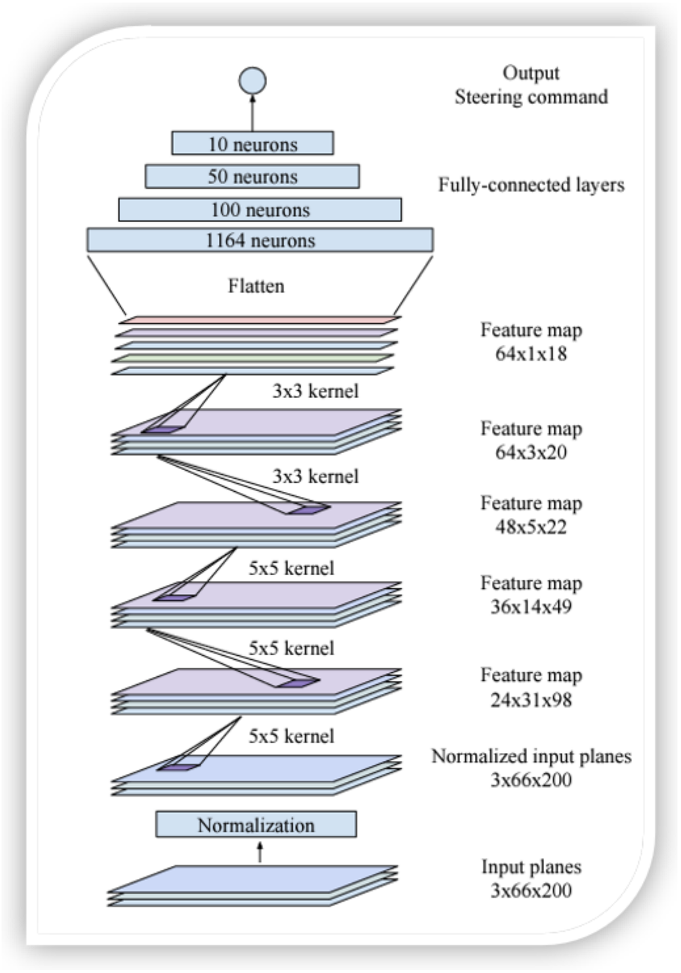


Figure 8.9: CNN Architecture [7]

Chapter 9

Integration & Deployment of Model in the Simulation

9.1 Model Integration

The trained model from the previous chapter 8.2.4 is integrated into the simulator environment using a Python script. Consistency between the training and prediction phases is maintained by employing the same preprocessing techniques in figure 8.5, thereby ensuring that the model receives the data in the expected format.

9.2 Real-time Communication

Real-time bi-directional communication between the simulator (client) and the Python script hosting the trained model (server) is established using Socket.IO. This technology allows the model and simulator to exchange real-time data such as images, speed, steering angles, and throttle values.

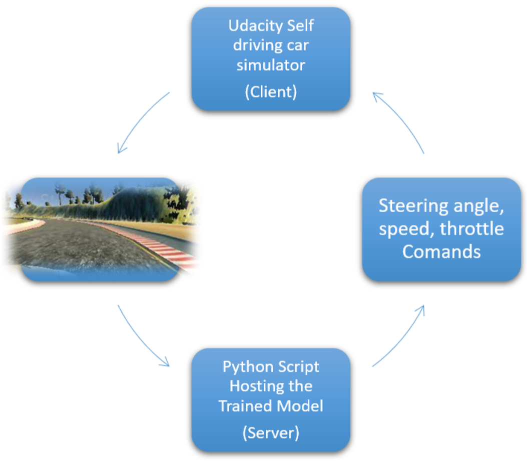

Udacity Self driving car simulator
(Client)
Steering angle, speed, throttle Comands
Python Script Hosting the Trained Model
(Server)

Figure 9.1: Validation Vs Training Data loss

Client-Server Communication:

Client (Simulator): The simulator acts as the client, continuously sending telemetry data to the server. This data includes the current speed of the car and images from the front-facing camera. The simulator also receives and acts upon control commands from the server, such as steering angle adjustments and throttle changes.
Server Side (Trained Model): The server running the trained model, receives the telemetry data from the client. It preprocesses the images to match the input shape and format used during the training, then predicts the appropriate steering angle using the loaded CNN model. The throttle is also calculated to control the speed, and both the steering angle and throttle are sent back to the simulator.

The code snippet below demonstrates the establishment of this communication:

```
@sio.on('telemetry')
def telemetry(sid, data):
    speed = float(data['speed'])
    image =
     ↪  Image.open(BytesIO(base64.b64decode(data['image'])))
    image = np.asarray(image)
    image = img_preprocess(image)
    image = np.array([image])
    steering_angle = float(model.predict(image))
    throttle = 1.0 - speed/speed_limit
    print('{} {} {}'.format(steering_angle, throttle, speed))
    send_control(steering_angle, throttle)
```

Figure 9.2: Here, the server listens for a 'telemetry' event and responds by processing the data and sending back control commands.

9.3 Control System

Throttle calculation is done to regulate the speed of the car based on the current speed and a predefined speed limit. A simple proportional control is implemented to maintain the speed within desired limits as illustrated in figure 9.2

The predicted steering angle and the calculated throttle are then packaged into a control command and sent back to the simulator. This communication is handled by the `send_control` function in our script, which emits the control data to the client:

```python
def send_control(steering_angle, throttle):
    sio.emit('steer', data = {
        'steering_angle': steering_angle.__str__(),
        'throttle': throttle.__str__()
    })
```

Figure 9.3: Function to send control commands to the simulator (client)

This efficient and responsive control system will enable the model to navigate the car through the simulation in real-time autonomously. Making it ready for testing.

9.4 Testing in the simulation

In our experimental validation of the model, we deployed a range of carefully chosen methodologies based on their relevance and efficacy in image classification tasks. First and foremost, the validation loss of our model stood at an impressive value of 0.045. This low value underscores the model's strong ability to generalize to unseen data, thus minimizing the error between the predicted outputs and the actual steering angles collected. The model employed Adam optimizer for efficient learning and (MSE) loss function for its regression problem of predicting the appropriate steering angles for a self-driving car based on the input image data.

In our experimental deployment of the model within the simulator, the bidirectional communication is established, and the autonomous vehicle model successfully navigated Track 1, exhibiting proficient driving capabilities and reinforcing the efficacy of our end-to-end learning approach in real time. However, certain steering inefficiencies were observed when challenged with the complexities of Track 2, particularly with its sharper turns. To counteract these challenges, targeted model optimizations were undertaken by optimizing the speed limit using a proportional controller applied to the car's speed inside the simulator to handle sharper turns more smoothly, resulting in enhanced decision-making latency and improved handling.

Chapter 10

Conclusion

To conclude, our exploration began by surveying the autonomy-level definitions and the current landscape of feasible applications and simulators for self-driving cars. Tools like CARLA, Airsim, CarSim, and Udacity offer platforms that mimic real-world scenarios, enabling us to test our autonomous driving systems in simulated environments. These tools are indispensable, ensuring that systems have undergone rigorous trials in controlled, reproducible conditions before any real-world testing.

We then delved into the various approaches to autonomous driving. The non-AI methods, represented by strategies like PID and MPC Control, provided a glimpse into the historical methods that laid the foundation for modern control systems. While robust in certain conditions, these methods do not scale well with the increasing complexity of real-world scenarios. On the other hand, the AI approach has been emerging as a dominant strategy, given its ability to learn from data, adapt to new situations, and handle the intricate complexities of driving.

The heart of this thesis lies in the Machine Learning segment, where the basics of algorithms we developed are covered through the theoretical concepts of supervised learning with linear and logistic regression, which paved the way for more advanced techniques like DNN introducing concepts like feed-forward neural mechanisms, back-propagation, and various gradient descent techniques. These networks, with their deep architectures, have the capability to capture intricate patterns in vast volumes of data. Then, CNN stood out as the best candidate for image data, which is crucial for autonomous driving. With specialized operations like convolution and pooling, CNNs have been instrumental in pushing the boundaries of what's possible in visual recognition tasks.

Our thesis concluded with our model's design, implementation, and testing. From collecting data and ensuring it's in the right format with image processing techniques to choosing the appropriate model and finally evaluating its performance, we ensured each step was undertaken with diligence and precision. The integration of this model into a simulation environment then became the final testament to its efficacy. With real-time communication and an integrated control system, we successfully enabled the car to navigate autonomously.

Bibliography

[1] Başargan, H. Driver and pedestrian trust analysis on integration of autonomous vehicles to infrastructure of turkey. *Alphanumeric Journal*, volume 7, 2019: pp. 25–36.

[2] Ynergy, P. PID Controller. Year. Available from: `https://plcynergy.com/pid-controller/`

[3] Perceptron: A Basic Neural Network Model for Deep Learning. `https://towardsai.net/p/l/perceptron-a-basic-neural-network-model-for-deep-learning`.

[4] Lin, C.; Chang, Q.; et al. A deep learning approach for MIMO-NOMA downlink signal detection. *Sensors*, volume 19, no. 11, 2019: p. 2526.

[5] Ng, D.; Feng, M. Medical Image Recognition: An Explanation and Hands-On Example of Convolutional Networks. *Leveraging Data Science for Global Health*, 2020: pp. 263–284.

[6] Alzubaidi, L.; Zhang, J.; et al. Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of big Data*, volume 8, 2021: pp. 1–74.

[7] Bojarski, M.; Del Testa, D.; et al. End to end learning for self-driving cars. arXiv 2016. *arXiv preprint arXiv:1604.07316*, volume 103, 2016.

[8] Bentley, J. *Autopia: The Future of Cars*. Atlantic Books, 2019.

[9] Wikipedia. Tesla Autopilot — Wikipedia, The Free Encyclopedia. 2023. Available from: `https://en.wikipedia.org/wiki/Tesla_Autopilot#cite_note-1`

[10] Audi. Audi AI: Traffic jam pilot. `https://magazine.audi.com.au/article/audi-ai-traffic-jam-pilot`.

[11] Singh, S.; Saini, B. S. Autonomous cars: Recent developments, challenges, and possible solutions. In *IOP Conference Series: Materials Science and Engineering*, volume 1022, IOP Publishing, 2021, p. 012028.

[12] Talebpour, A.; Mahmassani, H. S. Influence of connected and autonomous vehicles on traffic flow stability and throughput. *Transportation research part C: emerging technologies*, volume 71, 2016: pp. 143–163.

[13] Dosovitskiy, A.; Ros, G.; et al. CARLA: An open urban driving simulator. In *Conference on robot learning*, PMLR, 2017, pp. 1–16.

[14] Pretschner, A.; Broy, M.; et al. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering (FOSE'07)*, IEEE, 2007, pp. 55–71.

[15] Mnih, V.; Badia, A. P.; et al. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, PMLR, 2016, pp. 1928–1937.

[16] Shah, S.; Dey, D.; et al. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics: Results of the 11th International Conference*, Springer, 2018, pp. 621–635.

[17] Kendall, A.; Hawke, J.; et al. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 8248–8254.

[18] Doe, J. CarSim: Vehicle Dynamics Simulation Software. 2023. Available from: `https://www.carsim.com/products/carsim/index.php`

[19] Gelbal, Ş. Y.; Tamilarasan, S.; et al. A connected and autonomous vehicle hardware-in-the-loop simulator for developing automated driving algorithms. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2017, pp. 3397–3402.

[20] Joshi, A. Real-Time Implementation and Validation for Automated Path Following Lateral Control Using Hardware-in-the-Loop (HIL) Simulation. Technical report, SAE Technical Paper, 2017.

[21] Doe, J. Using Deep Learning to Predict Steering Angles. 2023. Available from: `https://medium.com/udacity/challenge-2-using-deep-learning-to-predict-steering-angles-f42004a36ff3`

[22] Smolyakov, M.; Frolov, A.; et al. Self-driving car steering angle prediction based on deep neural network an example of CarND udacity simulator. In *2018 IEEE 12th international conference on application of information and communication technologies (AICT)*, IEEE, 2018, pp. 1–5.

[23] Vilas Samak, C.; Vilas Samak, T.; et al. Control Strategies for Autonomous Vehicles. *arXiv e-prints*, 2020: pp. arXiv–2011.

[24] Russell, S. J. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[25] Grigorescu, S.; Trasnea, B.; et al. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, volume 37, no. 3, 2020: pp. 362–386.

[26] Hofmann, M.; Neukart, F.; et al. Artificial intelligence and data science in the automotive industry. *arXiv preprint arXiv:1709.01989*, 2017.

[27] MarketsandMarkets. Automotive Artificial Intelligence Market by Offering, Technology, Process, Drive Type, Application, Vehicle Type, and Region - Global Forecast to 2025. Year. Available from: `https://www.marketsandmarkets.com/PressReleases/automotive-artificial-intelligence.asp`

[28] Mehrotra, S.; Wang, M.; et al. Human-Machine Interfaces and Vehicle Automation: A Review of the Literature and Recommendations for System Design, Feedback, and Alerts. 2022.

[29] Debernard, S.; Chauvin, C.; et al. Designing human-machine interface for autonomous vehicles. *IFAC-PapersOnLine*, volume 49, no. 19, 2016: pp. 609–614.

[30] Chen, Z. Computer vision and machine learning for autonomous vehicles. *View at*, 2017.

[31] Russell, S.; Norvig, P. Artificial Intelligence A Modern Approach Third Edition, 2016.

[32] LeCun, Y.; Bengio, Y.; et al. Deep learning. *nature*, volume 521, no. 7553, 2015: pp. 436–444.

[33] What is Linear Regression? `https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-linear-regression/`.

[34] Logistic Regression in Machine Learning. `https://www.javatpoint.com/logistic-regression-in-machine-learning`.

[35] Vohwinkel, N. Linear models of regression. 2020.

[36] Science, T. D. How are Logistic Regression & Ordinary Least Squares Regression related? 2021. Available from: `https://towardsdatascience.com/how-are-logistic-regression-ordinary-least-squares-regression-related-1deab32d79f5`

[37] Vidhya, A. Understanding Gradient Descent Algorithm. 2021. Available from: `https://www.analyticsvidhya.com/blog/2021/03/understanding-gradient-descent-algorithm/`

[38] Wikipedia. Sigmoid function. 2023. Available from: `https://en.wikipedia.org/wiki/Sigmoid_function`

[39] Labs, V. A Guide to Cross-Entropy Loss. 2021. Available from: `https://www.v7labs.com/blog/cross-entropy-loss-guide`

[40] Cross entropy. `https://en.wikipedia.org/wiki/Cross_entropy`.

[41] Nielsen, M. A. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[42] Kumar, K. H. Computer vision and Machine Learning in Autonomous Vehicle. *Computing Department, Bournemouth University.*

[43] Geeks, G. D. Gradient Descent in Linear Regression. 2023. Available from: `https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/`

[44] in Linear Regression, G. D. Gradient Descent in Linear Regression. 2023. Available from: `https://copyassignment.com/gradient-descent-linear-regression/`

[45] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, volume 65, no. 6, 1958: p. 386.

[46] Wikipedia contributors. Perceptron — Wikipedia, The Free Encyclopedia. 2023, [Online; accessed 2-August-2023]. Available from: `https://en.wikipedia.org/wiki/Perceptron`

[47] Du, K.-L.; Leung, C.-S.; et al. Perceptron: Learning, generalization, model selection, fault tolerance, and role in the deep learning era. *Mathematics*, volume 10, no. 24, 2022: p. 4730.

[48] Szandała, T. Review and comparison of commonly used activation functions for deep neural networks. *Bio-inspired neurocomputing*, 2021: pp. 203–224.

[49] Anon. When to Use Softmax Activation. `http://www.mplsvpn.info/2017/12/when-to-use-softmax-activation.html`, 2017.

[50] Krizhevsky, A.; Sutskever, I.; et al. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, volume 60, no. 6, 2017: pp. 84–90.

[51] Hinton, G.; Deng, L.; et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, volume 29, no. 6, 2012: pp. 82–97.

[52] Khodadadi, A.; Ghandiparsi, S.; et al. A natural language processing and deep learning based model for automated vehicle diagnostics using free-text customer service reports. *Machine Learning with Applications*, volume 10, 2022: p. 100424.

[53] Goodfellow, I.; Bengio, Y.; et al. Deep feedforward networks. *Deep learning*, , no. 1, 2016.

[54] Grosse, R. Lecture 5: Multilayer Perceptrons. *inf. téc*, 2019.

[55] Ruder, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[56] Kingma, D. P.; Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[57] Graves, A.; Mohamed, A.-r.; et al. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, Ieee, 2013, pp. 6645–6649.

[58] Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, volume 79, no. 8, 1982: pp. 2554–2558.

[59] Elman, J. L. Finding structure in time. *Cognitive science*, volume 14, no. 2, 1990: pp. 179–211.

[60] Sutskever, I.; Vinyals, O.; et al. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, volume 27, 2014.

[61] Lipton, Z. C.; Berkowitz, J.; et al. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[62] Staudemeyer, R. C.; Morris, E. R. Understanding LSTM–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*, 2019.

[63] Wikipedia. ImageNet. 2023. Available from: `https://en.wikipedia.org/wiki/ImageNet`

[64] Worldwide Large Scale Visual Recognition Challenge Error Rates. 2023. Available from: `https://www.statista.com/statistics/808190/worldwide-large-scale-visual-recognition-challenge-error-rates/`

[65] Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, volume 36, no. 4, 1980: pp. 193–202.

[66] LeCun, Y.; Bottou, L.; et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, volume 86, no. 11, 1998: pp. 2278–2324.

[67] Shanagoda, S. Vehicle detection using faster regional convolutional neural network. *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, volume 8, no. 12, 2019: pp. 2701–2704.

[68] Yu, H.; Yang, L. T.; et al. Convolutional neural networks for medical image analysis: state-of-the-art, comparisons, improvement and perspectives. *Neurocomputing*, volume 444, 2021: pp. 92–110.

[69] Öztürk, G.; Köker, R.; et al. Recognition of vehicles, pedestrians and traffic signs using convolutional neural networks. In *2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, IEEE, 2020, pp. 1–8.

[70] Keras: The Python Deep Learning API. Available from: `https://keras.io/`

Chapter